Cyclone

A Safe Dialect of C

CMSC 631
Fall 2006

Credit where credit is due …

- Cyclone is a research language, the product of many collaborators:
  - Greg Morrisett, Yanling Wang (Harvard)
  - Dan Grossman (Washington)
  - Nikhil Swamy (Maryland)
  - Trevor Jim (AT&T)

1988? 2006?

- "In order to start copies of itself running on other machines, the worm took advantage of a buffer overrun...
- ...it is estimated that it infected and crippled 5 to 10 percent of the machines on the Internet."
- More than 15 years later, nearly half of CERT advisories involve buffer overruns, format string attacks, and similar low-level attacks.

The C Programming Language

- Critical software is often coded in C
  - device drivers, kernels
  - file systems, web servers, email systems
  - switches, routers, firewalls
- … most arguably because it is **low-level**
  - Control over data structure representations
  - Control over memory management
  - **Manifest cost**: good performance

Low-level, but unsafe

- Must bypass the type system to do even simple things (e.g., allocate and initialize an object)
- Libraries put the onus on the programmer to do the "right thing" (e.g., check return codes, pass in large enough buffer)
- For efficiency, programmers stack-allocate arrays of size K (is K big enough? does the array escape downwards?)
- Programmers assume objects can be safely recycled when they cannot, and fail to recycle memory when they should.
- It's not "fail-stop"-errors don't manifest themselves until well after they happen (e.g., buffer overruns.)

What about Java?

- Java provides safety in part via:
  - hidden data fields and run-time checks
  - automatic memory management
- Data representation and resource management are essential aspects of low-level systems

Cyclone

A safe, convenient, and modern language at the C level of abstraction

- Safe: memory safety, abstract types; fail-stop
- C-level: user-controlled data representation and resource management, easy interoperability, "manifest cost"
- Convenient: may need more type annotations, but work hard to avoid it
- Modern: add features to capture common idioms

"New code for legacy or inherently low-level systems"

Outline

- Status
- How Cyclone handles pointer errors
  - Spatial Errors
  - Temporal Errors
- Programming Experience
- Performance Analysis

Status

- >110K lines of Cyclone code
  - 80K compiler, libraries
  - 30K various servers, applications, device drivers
- gcc back-end (Linux, Cygwin, OSX, LEGO, …)
- User's manual, mailing lists, …
- Still a research vehicle (though winding down)

Projects using Cyclone

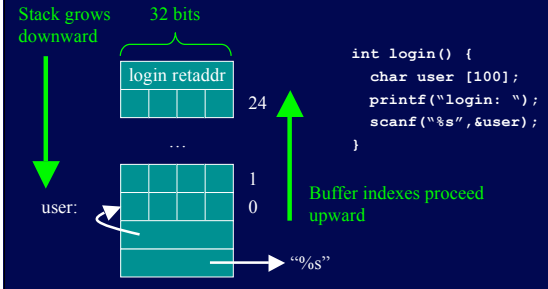
- Open Kernel Environment [Bos/Samwel, OPENARCH 02]
- MediaNet [Hicks et al, OPENARCH 03]
- RBClick [Patel/Lepreau, OPENARCH 03]
- STP [Patel et al., SOSP 03]
- FPGA synthesis [Teifel/Manohar, ISACS 04]
- O/S class at Maryland [2004-2005]

What is a C buffer overflow?

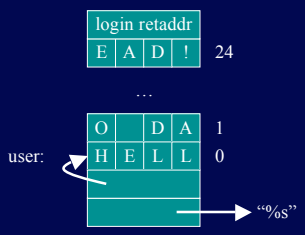
```
#include <stdio>

int login() {
  char user [100];
  printf("login: ");
  scanf("%s",&user);
  … // get password etc.
}
```

What happens if the user types In something that's more than 100 characters?

Calling scanf()

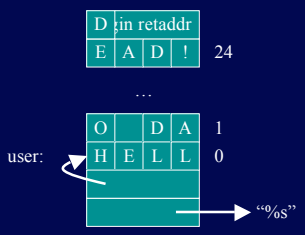


User types login



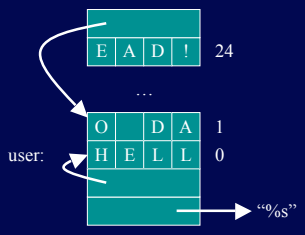

```
int login() {
  char user [100];
  printf("login: ");
  scanf("%s",&user);
}
```

101st character …




```
int login() {
  char user [100];
  printf("login: ");
  scanf("%s",&user);
}
```

Stack smashed!




```
int login() {
  char user [100];
  printf("login: ");
  scanf("%s",&user);
}
```

Can make the return address point anywhere, including into the buffer.

Abstraction-violating Attack

- Language and library abstractions may not be enforced
  - Array accesses, pointer dereferences, type casts, format strings "trusted" by the compiler
- Other attacks exploit this fact
  - Heap-based buffer overruns
  - Format string attacks

Two kinds of Pointer Errors

- Spatial
  - Dereferencing outside of a legal memory buffer, possibly at the wrong type
  - Abstraction-violating attacks in this category
- Temporal
  - Dereferencing a pointer after the pointed-to buffer has been freed

Preventing Spatial Errors

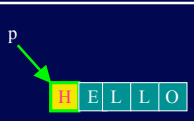
- Don't allow dereferencing a pointer unless compiler can prove it's safe
  - Often too conservative
- Prevent dereferencing with *dynamic* checks
  - May be able to eliminate some or all of these with static analysis, or programmer-provided type annotations
  - Safety first; then tune performance

Thin Pointers

A "thin" pointer (one word)

```
*p
p = NULL;
*p
```

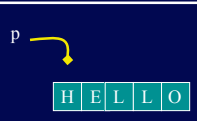
Thin Pointers



Only dereference permitted, no bounds check needed

```
*p
p = NULL;
*p
```

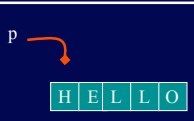
Thin Pointers



May be null

```
*p
p = NULL;
*p
```

Thin Pointers



Requires a null check if so

```
*p
p = NULL;
*p
```

Thin Pointers

Types and qualifiers for more flexibility and/or fewer checks

```
char *p;
char * @notnull p1;          // illegal: p1=NULL; p1=p
char * @notnull @numelts(6) p2;  // illegal: p2=p1
```

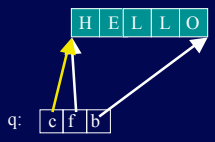
Thin Pointers

Shorthand

```
char *p;
char @ p1;
char @{6} p2;
```

Fat Pointers

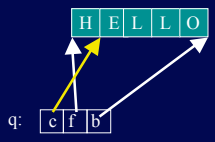
A "fat" pointer; has run time bounds: 3-word representation




```
q++;
q[0]
q--;
q--;
q[0]
```

Fat Pointers

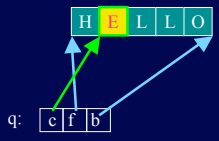
Pointer arithmetic OK




```
q++;
q[0]
q--;
q--;
q[0]
```

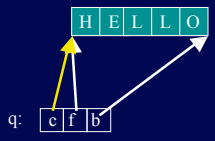
Fat Pointers

Bounds check on dereference




```
q++;
q[0]
q--;
q--;
q[0]
```

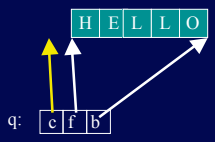
Fat Pointers




```
q++;
q[0]
q--;
q--;
q[0]
```

Fat Pointers

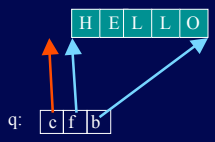
Dangling pointers OK …




```
q++;
q[0]
q--;
q--;
q[0]
```

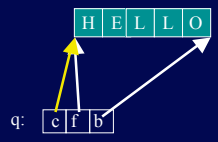
Fat Pointers

… caught on dereference




```
q++;
q[0]
q--;
q--;
q[0]
```

Fat Pointers

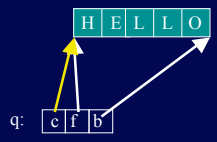



```
q++;
q[0]
q--;
q--;
q[0]
```

Types and qualifiers

```
char * @fat q;
```

Fat Pointers




```
q++;
q[0]
q--;
q--;
q[0]
```

Shorthand

```
char ? q;
```

Thin Pointer, Dynamic Bounds

len = 5

```
void foo(int len, char * @numelts(len) p) {
  for (int i = 0; i<len; i++)
    p[i] = …
}
```

Pointer Qualifier Summary

- @fat
  - rep. as a triple: {base, upper, curr}
  - supports all C pointer ops
  - but any dereference (may be) checked
- @notnull
  - Obviates null check (compiler must prove)
- @numelts(n)
  - Obviates bounds check (compiler must prove)
  - Can refer to dynamic lengths
- @zeroterm
  - Pointer is zero-terminated

Interfacing with libC

```
FILE *fopen(char * @notnull @zeroterm name,
            char * @notnull @zeroterm mode);

int putc(char, FILE * @notnull);

… fgets(char * @zeroterm @numelts(len),
        int len, FILE * @notnull);
```

Our **buildlib** tool easily generates platform dependent headers with signatures like these (programmer helps)

Temporal Errors

```
pt *add(pt *p, pt *q) {
  pt r;
  r->x = p->x + q->x;
  r->y = p->y + q->y;
  return &r;
}

void foo() {
  pt a = {1,2};
  pt b = {3,4};
  pt *c = add(&a, &b);
  c->x = 10;
}

typedef struct { int x; int y; } pt;
```

r's lifetime ends here!

so dereferencing c here can cause problems…

Preventing Temporal Errors

- Tracks object lifetimes by associating a *region* with each pointer:
  `int* @region(`r)`
  - A pointer can only be dereferenced while the region is still live. `int *`r` for short.
- Two basic kinds of regions
  - A lexical block (I.e., an activation record)
  - The heap (`H); has a global lifetime.

Simple Region Example

```
pt a = {1,2};

void foo() {
  pt b = {3,4};
  pt @`H aptr = &a;
  pt @`foo bptr = &b;
  addTo(&a, &b);
}
```

a lives in the heap region, so &a has type pt @`H.

b lives in the activation record of foo so &b has type pt @`foo.

region inference can figure out the regions, so the programmer doesn't have to write them

Region Polymorphism

```
void addTo<`r1,`r2>(pt *`r1 p, pt *`r2 q) {
  p->x += q->x;
  p->y += q->y;
}
```

addTo is parameterized by the regions for p and q.

This is standard parametric polymorphism:

addTo: ∀`r1. ∀`r2. (pt *`r1 × pt *`r2) → void

So this would go through...

```
pt *`H add<`r1,`r2>(pt *`r1 p, pt *`r2 q) {
  pt *r = malloc(sizeof(pt));
  r->x = p->x + q->x;
  r->y = p->y + q->y;
  return r;
}

pt a = {1,2};

void foo() {
  pt b = {3,4};
  pt *`H c = add<`H,`foo>(&a, &b);
  pt *`H d = add<`foo,`foo>(&b, &b);
  c->x = 10;
}
```

And this would be caught

```
pt *`H add<`r1,`r2>(pt *`r1 p, pt *`r2 q) {
  pt r;
  r.x = p->x + q->x;
  r.y = p->y + q->y;
  return &r;
}

pt a = {1,2};

void foo() {
  pt b = {3,4};
  pt *`H c = add<`H,`foo>(&a, &b);
  pt *`H d = add<`foo,`foo>(&b, &b);
  c->x = 10;
}
```

region of r is `add, not `H

On the other hand...

```
pt *`H add<`r1,`r2>(pt *`r1 p, pt *`r2 q) {
  p->x += q->x;
  p->y += q->y;
  return p;
}

pt a = {1,2};

void foo() {
  pt b = {3,4};
  pt *`H c = add<`H,`foo>(&a, &b);
  pt *`H d = add<`foo,`foo>(&b, &b);
  c->x = 10;
}
```

region of p is `r1, not `H

So we must be explicit

```
pt *`r1 add<`r1,`r2>(pt *`r1 p, pt *`r2 q) {
  p->x += q->x;
  p->y += q->y;
  return p;
}

pt a = {1,2};

void foo() {
  pt b = {3,4};
  pt *`H c = add<`H,`foo>(&a, &b);
  pt *`foo d = add<`foo,`foo>(&b, &b);
  c->x = 10;
}
```

What has to be written is thus:

```
pt *`r1 add(pt *`r1 p, pt * q) {
  p->x += q->x;
  p->y += q->y;
  return p;
}

pt a = {1,2};

void foo() {
  pt b = {3,4};
  pt * c = add(&a, &b);
  pt * d = add(&b, &b);
  c->x = 10;
}
```

The types say it all

```
pt * add (pt * p, pt * q);

pt *`r1 add (pt *`r1 p, pt * q);

pt *`r2 add (pt * p, pt *`r2 q);

char * @zeroterm `r
  fgets(char *{len} @zeroterm `r,
        int len,
        FILE @);
```

Dynamic Allocation

- How can we be sure data is live?
- Can use a GC, or safe manual techniques
- GC-based
  - Data allocated in heap is given region `H
    - Region is always live; dereferences are always safe
  - Conservative collector reclaims dead objects
  - Simple, but little control over performance
    - Potentially significant memory overheads
    - Pause times
    - May not be feasible in some environments (e.g., Linux)

Safe Manual Techniques

- Approach: generalize regions, track pointers
- New region kind: Arenas
  - Dynamic allocation, but all objects freed at once
- And/or impose aliasing restrictions
  - Can free individual objects using malloc/free or reference counting if aliasing is tracked
- When writing apps, use GC first, tune as necessary
  - Can result in significantly improved memory footprint and throughput

LIFO Arenas

- Dynamic allocation mechanism
- Lifetime of entire arena is scoped
  - At conclusion of scope, all data allocated in the arena is freed
  - Like a stack frame, but permits dynamic allocation
    - Useful when caller doesn't know how much memory needed by callee, but controls lifetime

LIFO Arena Example

```
FILE *infile = …
Image *i;
if (tag(infile) == HUFFMAN) {
  region<`r> h;   // region `r created
  struct hnode *`r huff_tree;
  huff_tree = read_tree(h,infile);
  // dynamically allocates with h
  i = decode_image(infile,huff_tree,…);
  // region `r deallocated upon exit of scope
} else …
```

Unique Pointers

- If object is known to have no aliases, it can be freed manually
  - Qualifier @aqual(\U), or \U for short
- An intraprocedural dataflow analysis
  - ensures that a unique pointer is not used after it is consumed (i.e. freed)
  - treats copies as destructive
    - one usable copy of a pointer to the same memory

Example

```
void foo() {
  int *\U x = malloc(sizeof(int));
  int *\U y = x;     // consumes x
  *x = 5;            // disallowed
  free(y);           // consumes y
  *y = 7;            // disallowed
}
```

Temporary Aliasing

- Problem: Non-aliasing too restrictive
- Partial solution: Allow temporary, lexically-scoped aliasing under acceptable conditions
  - Makes tracked pointers easier to use
  - Increases code reuse

Alias Construct

```
extern void f(int *`r x);      // `r any region

void foo() {
  int *\U x = malloc(sizeof(int));
  *x = 3;
  { alias <`r>int *`r y = x;       // `r fresh
    f(y);             // y aliasable, but x consumed
  }                                 // x unconsumed
  free(x);
}
```

With inference

```
extern void f(int * x);

void foo() {
  int *\U x = malloc(sizeof(int));
  *x = 3;
  f(x); // alias inserted here automatically
  free(x);
}
```

Reference-counted Pointers

- Aliasing qualifier \RC
  - pointed-to data have hidden count field
- Aliasing tracked as with unique pointers. Explicit aliasing/freeing via

```
`a *\RC`r alias_refptr(`a *\RC`r);
void drop_refptr(`a *\RC`r);
```

Interesting Combinations

- Tracked pointers can be freed manually, with `free` or `drop_refptr`, or automatically
  - Pointers into the heap freed by GC
  - Pointers into LIFO arenas freed at end of scope
    - Called a reap by Berger et al
- Can use tracked pointers to *keys* to permit arenas to have non-lexical lifetimes
  - Lifetime of arena corresponds to the lifetime of the key
  - Called dynamic arena

Summary

Region Variety	Allocation (objects)	Deallocation (what)	Deallocation (when)	Aliasing (objs)
Stack	static	whole region	exit of scope	ok
LIFO	dynamic			
Dynamic			manual	
Heap		single objects	GC	
Unique			manual	restricted
Refcounted				

Ensuring Uniformity and Reuse

- Many different idioms could be hard to use
  - Duplicated library functions
  - Hard-to-change application code
- We have solved this problem by
  - Using region types as a unifying theme
  - Region (and aliasing) polymorphism
    - E.g., functions independent of arguments' regions/aliasing
  - All regions can be treated as if lexical
    - Temporarily, under correct circumstances
    - Using alias and open (for dynamic arenas)

Some Application Experience

Boa	web server
Cfrac	Prime factorization
BetaFTPD	ftp server
Epic	image compression
Kiss-FFT	portable fourier transform
MediaNet	streaming overlay network
Linux Drivers	net, video, sound
CycWeb	web server
CycScheme	scheme interpreter

Application Characteristics

Program	Non-comment Lines of Code			Manual
	C	Cyc	Cyc (+manual)	mechs
Boa	5217	± 286 (5%)	± 98 (1%)	U
Cfrac	3143	± 183 (5%)	± 784 (25%)	UD
Betaftpd	1164	± 219 (18%)	± 238 (22%)	UR
Epic	2123	± 218 (10%)	± 117 (5%)	UL
KissFFT	453	± 74 (16%)	± 25 (5%)	U
8139too	1972	± 971 (49%)	± 312 (14%)	UD
i810_audio	2598	± 1500 (57%)	± 318 (10%)	RD
pwc	3755	± 1373 (36%)	± 1024 (26%)	RD
MediaNet		8715	± 320 (4%)	URLD
CycWeb			667	U
CycScheme			2523	ULD

U = unique pointers R = ref-counted pointers
L = LIFO regions D = dynamic arenas

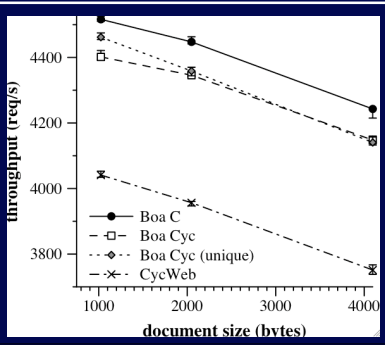
Experimental Measurements

- Platform
  - Dual 1.6 GHz AMD Athlon MP 2000
    - 1 GB RAM
    - Switched Myrinet
  - Linux 2.4.20 (RedHat)
- Software
  - C code: gcc 3.2.2
  - Cyclone code: cyclone 0.9
  - GC: BDW conservative collector 6.2α4
  - malloc/free: Lea allocator 2.7.2

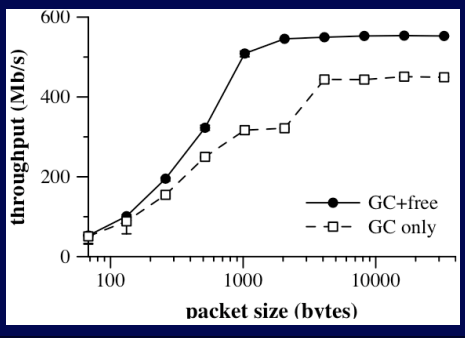
Bottom Line

- CPU time
  - I/O bound applications have comparable performance
    - All applications: at most 60% slowdown
  - GC has little impact on elapsed time
    - MediaNet is the exception
- Memory usage
  - Using GC requires far more memory than manual
  - Cyclone manual techniques approach footprint of C original

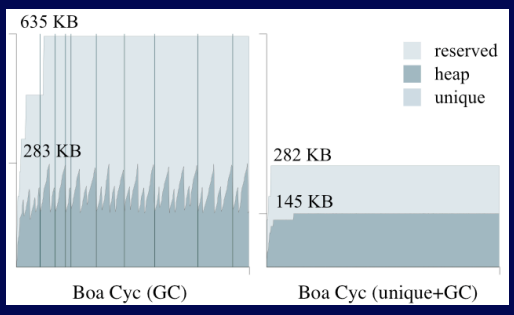
Throughput (Webservers)



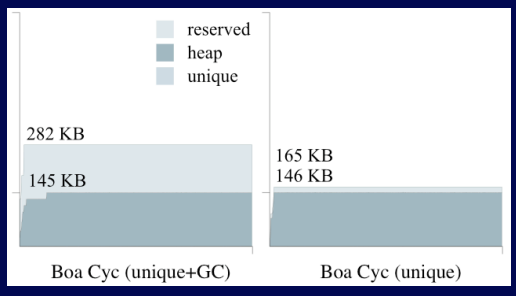
Throughput (MediaNet)



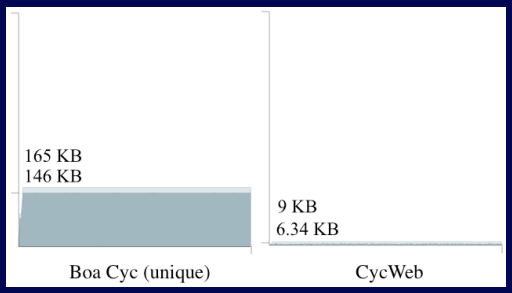
Memory Usage (Web)



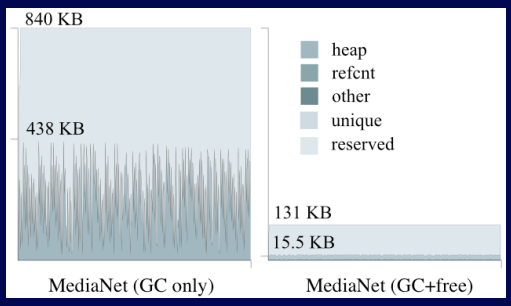
Memory Usage II (Web)



Memory Usage III (Web)



Memory Usage (MediaNet)



(4 KB packets)

Other Apps (C vs. Cyc GC)

Test	C		Cyclone GC	
	Time	Mem	Time	Mem
Epic	0.70	12.5M	1.11 (1.61)	22.3M (1.78)
KissFFT	1.33	394K	1.40 (1.05)	708K (1.80)
Betaftpd	4.00	6.2K	4.00 (1.0)	192K (30.1)
Cfrac	8.75	284K	15.23 (1.74)	1.44M (5.19)
8139too	334	27.7K		

Other Apps (Cyc GC vs. no GC)

Cyclone GC		Cyclone Manual	
Time	Mem	Time	Mem
1.11 (1.61)	22.3M (1.78)	1.11 (1.61)	12.5M (1.0)
1.40 (1.05)	708K (1.80)	1.41 (1.06)	392K (0.99)
4.00 (1.0)	192K (30.1)	4.00 (1.0)	8.2K (1.32)
15.23 (1.74)	1.44M (5.19)	14.53 (1.66)	706K (2.49)
		333(0.99)	31.8K (1.14)

Things I didn't talk about

- Modern language features too
  - Tagged unions and data types
  - Pattern matching
  - Exceptions
  - Allocation with **`new`**
- Porting tool
- Lots of libraries

Related Work: making C safer

- Compile to make dynamic checks possible
  - Safe-C [Austin et al.], RTC [Yong/Horwitz], ...
  - Purify, Stackguard, Electric Fence, ...
  - CCured [Necula et al.]
    - performance via whole-program analysis
    - less user burden
    - less memory management, single-threaded
- Control-C [Adve et al.] weaker guarantee, less burden
- SFI [Wahbe, Small, ...]: sandboxing via binary rewriting

Related Work: Checking C

- Model-checking C code (SLAM, BLAST, …)
  - Leverages scalability of MC
  - Key is automatic building and refining of model
  - *Assumes* (weak) memory safety
- Lint-like tools (Splint, Metal, PreFIX, …)
  - Good at reducing false positives
  - *Cannot* ensure absence of bugs
  - Metal particularly good for user-defined checks
- Cqual (user-defined qualifiers, lots of inference)

Better for unchangeable code or user-defined checks (i.e., they're complementary)

Related work: higher and lower

- Adapted/extended ideas:
  - polymorphism [ML, Haskell, …]
  - regions [Tofte/Talpin, Walker et al., …]
  - safety via dataflow [Java, …]
  - existential types [Mitchell/Plotkin, …]
  - controlling data representation [Ada, Modula-3, …]
- Safe lower-level languages [TAL, PCC, …]
  - engineered for machine-generated code
- Vault: stronger properties via restricted aliasing

Future Work

- Tracked pointers can be painful; want
  - Better inference (e.g. for alias)
  - Richer API (restrict; autorelease)
- Prevent leaks
  - unique and reference-counted pointers
- Specified aliasing
  - for doubly-linked lists, etc.
- Concurrency

Conclusions

- High degree of control, *safely:*
- Sound mechanisms for low-level control
  - Checked pointers for spatial errors
  - Variety of techniques for temporal errors
    - Region-based vs. object-based deallocation
    - Manual vs. automatic reclamation
- Region- and alias-annotated pointers within a coherent framework
  - Scoped regions unifying theme (alias,open)
  - Polymorphism, for code reuse

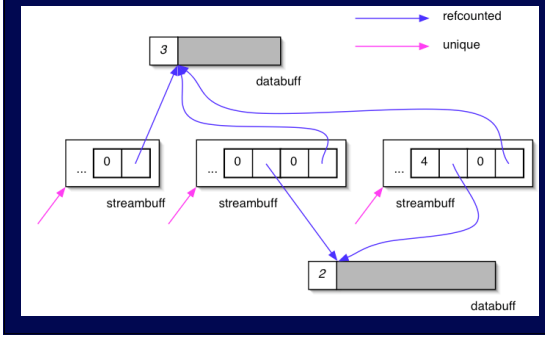
More Information

- Cyclone homepage

  http://cyclone.thelanguage.org

- Has papers, benchmarks, distribution

MediaNet Data structures



Other Details

- Can store unique pointers in non-unique containers
  - Must use "swap" operator to extract them, to ensure soundness of analysis.
- Alias construct
  - Like Walker/Watkins (and Wadler) `let!`
  - Fresh region is crucial to soundness - normal typing machinery ensures that aliased pointers will not escape the scope.

Sharing Unique Pointers

- Can we have unique pointers in non-unique containers? Options:
  - Don't allow them; norm in linear type systems
  - Track sharable containers
    - E.g., guarded types in Vault; restricts container aliasing, and may not be thread-safe
  - Swap out unique pointers

Swap to the rescue

```
int *`U g = NULL; // cannot access directly
void init(int v) {
  int *`U x = malloc(sizeof(int));
  *x = v;
  g :=: x; // atomically swap *x with *g
  free(x);
}
```

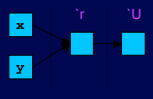
Tying it together: Polymorphism

- Goal: avoid writing slightly different versions of the same functions
  - One for unique pointers, one for lexically-scoped region pointers, etc.
- Approach: parametric polymorphism, differentiated by kinds.
  - Need to differentiate between generic functions that operate on alias-restricted pointers and freely-aliasable ones.

Unique-path access restriction

Invariant: Only ever one (unique) path by which to access a unique object at any program point

```
void f(int *`U*`r x, int *`U*`r y) {
  free(*x);
  **y = 1; // error if *x == *y
}
```



Dynamic Arenas

- Unique and reference-counted objects can be freed at any time
- Also want to be able to free an entire arena at any time
  - Not just at the conclusion of a scope; I.e. want separate region allocation and deallocation functions.

Dynamic Arena Approach

- Define a *key* as a unique or reference-counted pointer to a region handle
- Key lifetime corresponds with region lifetime
  - Freeing the key frees the entire region
- *Open* a region to use it within a lexical scope
  - makes region live within scope
  - consumes the key until done

Dynamic Arena Example

```
void f(struct DynamicRegion<`r> *`U k,
       int *`r x)
{
  { region r = open k; // `r live, k consumed
    *x = 42;
    bar(r,x);
  } // `r no longer live, k unconsumed
  free_ukey(k); // frees the key and the region
}
```

Dynamic Regions API

- **`struct DynamicRegion<`r::R> *`k::TR`**
  - Pointer to a container (in region `k) for a dynamic region `r. Called a *key*.
- **`new_ukey()`**
  - Returns a key, where `r is fresh (wraps in an existential), and `k is the unique region.
- **`new_rckey()`**
  - Similar, but `k is the ref-counted region.

Dynamic Regions API

- **`region r = open k;`**
  - Given **k** points to a **DynamicRegion<`r>**, makes `r live in the ensuing scope (with handle **r**), temporarily consuming **k**
- **`free_ukey(ukey)`**
  - Frees the key and the region it contains
- **`new_rckey(rckey)`**
  - Decrements the reference count on the key, freeing it and the region if it goes to zero

Disadvantages

- Programmers have to make the lifetimes of objects explicit:
  - must put regions on return types
  - must allocate objects within regions
  - must pass region handles around.
- Regions lifetimes must be LIFO
  - No easy way to handle objects with overlapping, non-nested lifetimes
- Constructing a region expensive if only allocates a small number of objects

Cyclone: where we stand

- Cyclone compiler
  - ~100KL of Cyclone code
  - Bulk is the type-checker and dataflow analyses
  - Straightforward translation to C
  - Available for many architectures (Linux, BSD, Irix, Cygwin, Sparc, etc.)
- Ports
  - Libc and other libs (sockets, XML, lists, and more)
  - bison, flex, web server, cfrac, grobner, NT device driver … (~40KL total)
  - Typically differ from original C by 5-15%

Growable Regions

```
typedef struct List<`r> {
  int head;
  struct List<`r> *`r tl;
} *`r list_t<`r>;

void foo(unsigned int i) {
  region<`d> h {
    list_t<`d> x = NULL, temp;
    for (; i != 0; i++) {
      temp = rmalloc(h, sizeof(struct List));
      temp->head = i; temp->tl = x;
      x = temp;
    };
    process(h,x);
  }
}
```

Growable Regions

```
typedef struct List<`r> {
  int head;
  struct List<`r> *`r tl;
} *`r list_t<`r>;

void foo(unsigned int i) {
  region<`d> h {
    list_t<`d> x = NULL, temp;
    for (; i != 0; i++) {
      temp = rmalloc(h, sizeof(struct List));
      temp->head = i; temp->tl = x;
      x = temp;
    };
    process(h,x);
  }
}
```

The list structure is parameterized by a region

Growable Regions

```
typedef struct List<`r> {
  int head;
  struct List<`r> *`r tl;
} *`r list_t<`r>;

void foo(unsigned int i) {
  region<`d> h {
    list_t<`d> x = NULL, temp;
    for (; i != 0; i++) {
      temp = rmalloc(h, sizeof(struct List));
      temp->head = i; temp->tl = x;
      x = temp;
    };
    process(h,x);
  }
}
```

The "region<`d> h{...}" introduces a new dynamic region `d with handle h.

Growable Regions

```
typedef struct List<`r> {
  int head;
  struct List<`r> *`r tl;
} *`r list_t<`r>;

void foo(unsigned int i) {
  region<`d> h {
    list_t<`d> x = NULL, temp;
    for (; i != 0; i++) {
      temp = rmalloc(h, sizeof(struct List));
      temp->head = i; temp->tl = x;
      x = temp;
    };
    process(h,x);
  }
}
```

Objects can be allocated into region `d using rmalloc(h,...) where h is a handle for the region.

Unlike the stack, any number of objects can be placed in the region.

Growable Regions

```
typedef struct List<`r> {
  int head;
  struct List<`r> *`r tl;
} *`r list_t<`r>;

void foo(unsigned int i) {
  region<`d> h {
    list_t<`d> x = NULL, temp;
    for (; i != 0; i++) {
      temp = rmalloc(d, sizeof(struct List));
      temp->head = i; temp->tl = x;
      x = temp;
    };
    process(h,x);
  }
}
```

Since this function is given the handle for `d, it can allocate objects in there too, or return results in that region.

Growable Regions

```
typedef struct List<`r> {
  int head;
  struct List<`r> *`r tl;
} *`r list_t<`r>;

void foo(unsigned int i) {
  region<`d> h {
    list_t<`d> x = NULL, temp;
    for (; i != 0; i++) {
      temp = rmalloc(h, sizeof(struct List));
      temp->head = i; temp->tl = x;
      x = temp;
    };
    process(h,x);
  }
}
```

The entire region is deallocated at the end of its scope.

So Far: Advantages

- Type system makes sure that:
  - can't dereference a pointer to a freed object.
  - can't forget to free a region.
  - supports some dangling pointers.
- Runtime system ensures that:
  - region and object allocation are constant time.
  - region deallocation is constant time – and faster than individually free'ing the objects.
- So the approach is quite attractive for real-time systems when compared to GC.

Example

```
struct conn *\RC cmd_pasv(struct conn *\RC c) {
  struct ftran *\RC f;
  int sock = socket(...);
  f = alloc_new_ftran(sock,alias_refptr(c));
  c->transfer = alias_refptr(f);
  listen(f->sock, 1);
  f->state = 1;
  drop_refptr(f);
  return c;
}
```